dotCall64: An R package providing an efficient interface to compiled C, C++, and Fortran code supporting long vectors

Gerber, F ; Mösinger, K ; Furrer, Reinhard

Abstract: The R package dotCall64 provides an enhanced version of the foreign function interface (FFI) to call compiled C, C++, and Fortran code from the software environment R. It allows users to integrate compiled code without using complex application programming interfaces (APIs), such as the C API of R. Moreover, dotCall64 supports long vectors having more than elements and implements a mechanism to avoid unnecessary copies of R objects. Therefore, dotCall64 facilitates making existing C, C++, and Fortran libraries accessible for R and is particularly useful for applications involving long vectors.

DOI: https://doi.org/10.1016/j.softx.2018.06.002

SoftwareX 7 (2018) 217–221

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

dotCall64: An R package providing an efficient interface to compiled C, C++, and Fortran code supporting long vectors

F. Gerber [a,*], K. Mösinger [a], R. Furrer [a,b]

[a] Department of Mathematics, University of Zurich, Switzerland
[b] Department of Computational Science, University of Zurich, Switzerland

ARTICLE INFO

Article history:
Received 12 January 2018
Received in revised form 14 June 2018
Accepted 17 June 2018

Keywords:
Foreign function interface
64-bit
Large datasets

ABSTRACT

The R package dotCall64 provides an enhanced version of the foreign function interface (FFI) to call compiled C, C++, and Fortran code from the software environment R. It allows users to integrate compiled code without using complex application programming interfaces (APIs), such as the C API of R. Moreover, dotCall64 supports long vectors having more than $2^{31} - 1$ elements and implements a mechanism to avoid unnecessary copies of R objects. Therefore, dotCall64 facilitates making existing C, C++, and Fortran libraries accessible for R and is particularly useful for applications involving long vectors.

Software metadata

Current code version	v0.9-5.2
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2018_56
Legal Code License	GPL-3
Code versioning system used	git https://git.math.uzh.ch/reinhard.furrer/dotCall64
Software code languages, tools, and services used	R, C, openMP
Compilation requirements, operating environments	R ($\geq$ v3.1)
If available Link to developer documentation/manual	https://cran.r-project.org/web/packages/dotCall64/dotCall64.pdf
Support email for questions	florian.gerber@math.uzh.ch

1. Motivation and significance

The open-source programming language R is a popular software environment for statistical computing (https://www.r-project.org/). Its interpreted character makes R a convenient front-end for a wide range of applications. Although R provides a rich infrastructure, it can be advantageous to extend R programs with compiled code written in C, C++ or Fortran [1]. Reasons for such an extension are the access to new and trusted computations, the increase in computational speed, and the object referencing capabilities [2].

R provides two types of interfaces to call compiled code [3]. First, the *foreign function interface* (FFI) provides the R functions *.C()* and *.Fortran()*. This interface allows the compiled code to read and modify atomic R vectors, which are exposed as the corresponding C, C++ and Fortran types, respectively. Thus, no R specific adaptations of the compiled code are required, making it favorable for embedding C, C++ and Fortran code that is not specifically designed for R. Second, the *modern interfaces to C/C++ code* (MIC) feature the R functions *.Call()* and *.External()*. They enable accessing, modifying, and returning R objects from C using the C API of R [4]. This is convenient when C code is specifically written to be used with R. In that case, the C API serves as a glue between R and C, providing some R functionality and control over copying R objects on the C level. However, it requires the user to learn the C API of R. Especially, when an R interface is built on top of existing compiled code this constitutes an additional effort. Moreover, R has no Fortran API, and hence, the MIC are inconvenient to embed Fortran code into R.

On top of these interfaces, R packages exist that simplify the integration of compiled code into R. One such R package is inline [5], which allows the user to dynamically define R functions and S4 methods with inlined compiled code. Other examples are Rcpp [6–8] and its extensions RcppArmadillo [9,10], RcppEigen [11,12], RcppParallel [13], and Rcpp11 [14], which greatly simplify the

* Corresponding author.
E-mail address: florian.gerber@math.uzh.ch (F. Gerber).

https://doi.org/10.1016/j.softx.2018.06.002

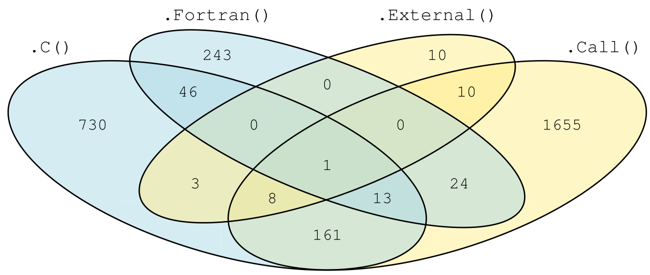

Fig. 1. Number of R packages on CRAN (https://cran.r-project.org/) using the foreign function interface (blue) and the modern interfaces to C/C++ code (yellow) to interface C, C++ or Fortran code (as of 2018-01-09). Note that from the 1′872 R packages using *.Call()* 1′321 (70%) link to Rcpp. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

extension of R with C++ code. Similar to the MIC, the Rcpp package family is designed to extend R with compiled code that is specifically written for that purpose.

In the remainder of this article, we focus on the intention to embed compiled code into R without using an additional C or C++ API. The natural solution for that task seems to be the FFI and about 40% of the packages on CRAN embedding compiled C, C++ or Fortran code use that interface (Fig. 1). An example of such an R package is the SPArse Matrix package spam [15–17] (prior to version 2), which is built around the Fortran library SPARSKIT [18]. Here, the R function *.Fortran()* from the FFI seems to be suitable. Conversely, using the MIC is also possible but requires adding an additional layer of C code to enable communication between R and the compiled Fortran code. However, using the FFI is also not satisfying, since it lacks flexibility and performance, as also stated in its help page: "These functions [*.C()* and *.Fortran()*] can be used to make calls to compiled C and Fortran 77 code. Later interfaces are '.Call' and '.External' which are more flexible and have better performance." Two of the missing features of the FFI are:

- a mechanism to avoid unnecessary copies of R objects,
- support of long vectors.

The former is the reason for the lower performance of the FFI compared to the MIC. Since the FFI does not allow R vectors to be passed to compiled code by reference (without copying), it is especially impractical for big data application. The missing features of the FFI motivated the development of the R package dotCall64, which is presented in this article.

To understand the main difficulty associated with passing long R vectors to compiled code it should be noted that R uses doubles to index long vectors, whereas compiled code naturally uses 64-bit integer (more details are given in Section S1 of the supplementary material). Therefore, passing indices of long vectors from R to compiled code involves castings (type conversions), which are fully automated and parallelized by the proposed interface. Conversely, one could think of using the 64-bit integer class provided by the R package bit64 [19] to avoid castings. However, that class cannot be used to index long R vectors, and hence, does not help to avoid castings.

2. Software description

The R package dotCall64 [20] contains R and C source code, documentation, examples, unit tests implemented with the R package testthat [21], and R scripts for performance measurements.

2.1. Software functionalities

The main R function of dotCall64 is *.C64()*, which is an enhanced replacement of the FFI and equally easy to use. The syntax of *.C64()* resembles that of the function *.C()*, and both functions have common arguments as shown in Table 1. Further details are also given in the reference manual [20].

Table 1
Arguments and default values of the R function *.C()* from the FFI and *.C64()* from dotCall64. The depreciated arguments of *.C()* are marked with "*".

.C()		*.C64()*	
arguments	*defaults*	*arguments*	*defaults*
.NAME		*.NAME*	
		SIGNATURE	
...		...	
		INTENT	*NULL*
NAOK	*FALSE*	*NAOK*	*FALSE*
**DUP*	*TRUE*		
PACKAGE		*PACKAGE*	""
**ENCODING*			
		VERBOSE	*getOption("dotCall64.verbose")*

Table 2
Supported *SIGNATURE* arguments of *.C64()* and the corresponding C, C++, Fortran, and R data types. The column "cast" indicates whether casting is necessary.

SIGNATURE	C, C++ type	Fortran type	R type	cast
"double"	*double*	*double precision*	*double*	no
"integer"	*integer*	*integer (kind = 4)*	*integer*	no
"int64"	*int64_t*	*integer (kind = 8)*	*double*	yes

The required arguments of *.C64()* are:

- *.NAME* The name of the compiled C, C++ function or Fortran subroutine.
- ... Up to 65 R vectors to be accessed by the compiled code.
- *SIGNATURE* A character vector of the same length as the number of arguments of the compiled function (also called subroutines for Fortran). Each string specifies the signature of one such argument. Accepted signatures are *"integer"*, *"double"*, and *"int64"*. The R, C, C++, and Fortran types corresponding to these specifications are given in Table 2.

With that, the following call to the compiled C function *void get_c(double input, int index, double output)* using *.C()* can be replaced by its *.C64()* counterpart. Therefore, for example,

```
R> .C("get_c", input = as.double(1:10), index = as.integer(9),
      output = double(1))
```

becomes

```
R> .C64("get_c", SIGNATURE = c("double", "integer", "double"),
        input = 1:10, index = 9, output = 0)
```

While more detailed code examples are given later, this is enough to highlight some features of *.C64()*. First, *.C64()* requires the additional argument *SIGNATURE* specifying the argument types of the compiled function. In return, it coerces the provided R vectors to the specified signatures making the *as.double()* and *as.integer()* statements unnecessary. Second, all provided arguments can be long vectors. Third, if one of the arguments of the compiled function is a 64-bit integer (*int64_t* in the case of C, C++ functions, and *integer (kind = 8)* types for Fortran subroutines), it is enough to set the corresponding *SIGNATURE* argument to *"int64"* to successfully evaluate the function. That is, *.C64()* does the necessary *double* to 64-bit integer and 64-bit integer to *double* castings before and after evaluating the compiled code, respectively. See Section S1 of the supplementary material for details on the long vector support of R, which explain why the mentioned castings are necessary.

Additional arguments of *.C64()* are the following:

- *INTENT* A character vector of the same length as the number of arguments of the compiled function. Each string specifies the intent of one such argument. Accepted intents are *"rw"* (read and write), *"r"* (read), and *"w"* (write).
- *NAOK* A logical flag specifying whether the R vectors passed though "..." are checked for missing and infinite values.
- *PACKAGE* A character vector of length one restricting the search path of the compiled function to the specified package.
- *VERBOSE* If *0* (default), no warnings are printed. If *1* and *2*, then warnings for tuning and debugging purposes are printed.

The argument *INTENT* influences the copying of R vectors and can be seen as an enhanced version of the depreciated *DUP* argument of *.C()*. By default, all intents are set to "read and write" implying that the compiled code receives pointers to copies of the R vector given to "...". This behavior is desirable when the compiled function reads the corresponding R vectors and modifies (writes to) them. For arguments of the compiled function that are only read and not modified, the intent can be set to "read". With that, the compiled code receives pointers to the corresponding R vectors itself. While this avoids copying, it is absolutely necessary that the compiled code does not alter these vectors, as this corrupts the corresponding R vectors in the current R session. Moreover, *.C64()* provides an efficient way to allocate zero initialized vectors, which are often used to return results of compiled functions. To achieve that, the intent "write" has to be specified and the corresponding R vectors passed to "..." have to be of class *"vector_dc"* (short for "zero initialized vector allocated by dotCall64"). R objects of that class contain information on the type and length, which is then used by *.C64()* to allocate the corresponding vectors. Thus, copying zero initialized vectors is avoided. Vectors of class *"vector_dc"* can be constructed with the R function *vector_dc()*, taking the same arguments as *vector()* from the base R package. For example, instead of passing the R vector *vector(mode = "numeric", length = 8)*, the following R object should be passed.

```
R> vector_dc(mode = "numeric", length = 8)
$mode
[1] "numeric"

$length
[1] 8

attr(,"class")
[1] "vector_dc" "list"
```

Note that specifying the suitable intent may reduce computation time by avoiding unnecessary copying of R vectors and by avoiding unnecessary *double* to 64-bit *integer* and 64-bit *integer* to *double* castings for *SIGNATURE = "int64"* type arguments. More details on performance relevant arguments are given in Section S2 of the supplementary material and the package manual [20].

2.2. Software architecture

The function *.C64()* uses the function *.External()* from the MIC to directly pass all provided arguments to the C function *dC64()*. After basic checks of the provided arguments *dC64()* proceeds as schematized in Fig. 2. Note that the flowchart shows the procedure for the case in which the compiled function has only one argument. Otherwise, *dC64()* repeats the scheme for all arguments.

One aspect to highlight is the castings of R vectors for *SIGNATURE = "int64"* arguments. For such arguments, the *double* to *int64_t* casting is done for the intents "read and write" and "read"; see the boxes labeled with (a). In that case, duplication is not necessary, as the implemented casting allocates a new vector anyway. The back-casting from *int64_t* to *double* is only done for the intents "read and write" and "write"; see the box labeled with (b).

Moreover, an argument of *SIGNATURE* different from *"int64"* with intent "read and write" is duplicated in any case; see boxes labeled with (c). If the intent is "read", it is not duplicated, and if the intent is "write", the argument is only duplicated when it has a reference status different from 0. R vectors increase their reference status when they are passed to an R function, and therefore, *.C64()* cannot rely on that mechanism. Instead, a safe way to avoid copying a zero initialized vector is to pass an R object of class *"vector_dc"*.

As casting is an expensive operation in terms of computational time, we distribute this task to multiple threads using openMP, if available [22,23]. Note that the number of used threads can be controlled with the R function *omp_set_num_threads()* from the package OpenMPController [24]. The package dotCall64 can also be compiled without the openMP feature by removing the flag *$(SHLIB_OPENMP_CFLAGS)* in the *src/Makevars* file of the source code.

3. Illustrative example

We showcase *.C64()* from the R package dotCall64 by using it to interface an example C function. A Fortran version of the example is provided in Section S3 of the supplementary material. A direct comparison of *.C64()* and *.C()* from the FFI shows some of the limitations of the FFI and that it is straight forward to overcome these with *.C64()*. Moreover, the similarities and differences in the syntax become visible. We consider the example C function defined next.

```
R> writeLines("
void get_c(double *input, int *index, double *output) {
    output[0] = input[index[0] - 1];
}", con = "get_c.c")
```

get_c() takes the arguments *input* (*double*), *index* (*integer*), and *output* (*double*) and writes the element of *input* at the position specified with *index* to *output*. We compile the function on the command line using *R CMD SHLIB get_c.c*. The resulting dynamic shared object (*get_c.so* on our Linux platform) must be loaded into R before the compiled function can be called. Note that, in the following R code, the extension of the shared object is replaced with *.Platform$dynlib.ext* to make the code platform independent.

```
R> dyn.load(paste0("get_c", .Platform$dynlib.ext))
```

One can use *.C()* from the FFI to call the C function *get_c()*. Note the use of the R functions *as.double()* and *as.integer()*, which ensure that the types of the passed R vectors match the signature of *get_c()*.

```
R> .C("get_c", input = as.double(1:10), index = as.integer(9),
      output = double(1))$output
[1] 9
```

Next, we try to use the same call with a long vector *x_long* passed to the argument *input* of *get_c()*. Note that the following R code requires up to 32 GB of free memory to run.

```
R> x_long <- double(2^31); x_long[9] <- 9; x_long[2^31] <- -1
R> .C("get_c", input = as.double(x_long), index = as.integer(9),
      output = double(1))$output
Error: long vectors (argument 1) are not supported in .C
```

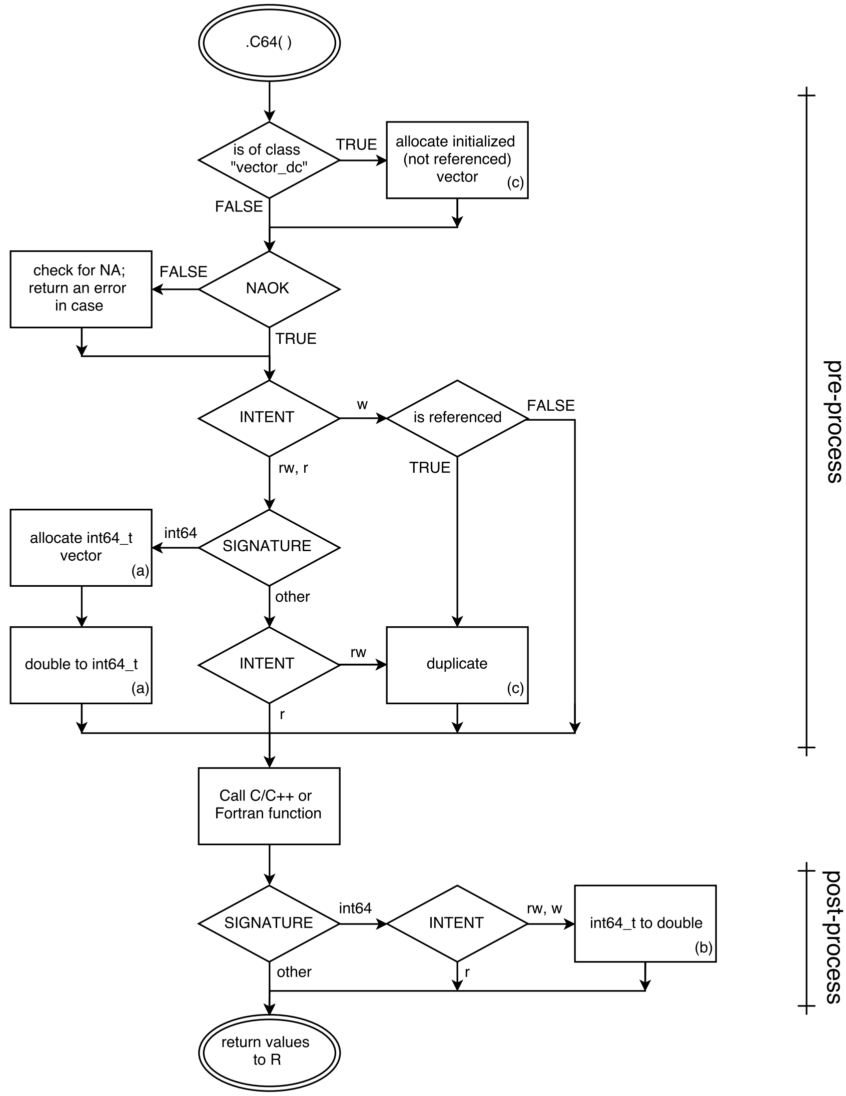

Fig. 2. Flowchart of the involved processes when using *.C64()* to call a compiled function with one argument. In the pre-process phase, the provided R vector passed through "..." is checked and prepared according to the arguments *NAOK*, *SIGNATURE*, and *INTENT*. Then, the compiled function specified with the argument *.NAME* is called. Finally, the vector is back-cast in the post-process phase if necessary.

As expected, *.C()* throws an error because it does not support long vectors. The error can be avoided by replacing *.C()* with *.C64()*. This allows the evaluation of the C function *get_c()* with the long vector *x_long*. Additionally, *.C64()* requires the argument *SIGNATURE* encoding the signatures of the arguments of *get_c()*. This information is used to coerce all provided R vectors to the specified signatures. Thus, it is no longer necessary to reassure that the types of the passed R vectors match the signature of the compiled function.

```
R> install.packages("dotCall64")
R> library("dotCall64")
R> .C64("get_c", SIGNATURE = c("double", "integer", "double"),
      input = x_long, index = 9, output = double(1))$output
[1] 9
```

In contrast to the call using *.C()*, the ninth element of the long vector *x_long* is returned. However, the argument *index* of *get_c()* is of type *int* (a 32-bit integer), and hence, elements at positions beyond $2^{31} - 1$ cannot be extracted. To overcome this, we adapt the definition of the C function *get_c()* and replace the *int* type in the declaration of the argument *index* with the *int64_t* type, which is defined in the C header file *stdint.h*.

```
R> writeLines("
#include <stdint.h>
void get64_c(double *input, int64_t *index, double *output) {
    output[0] = input[index[0] - 1];
}", con = "get64_c.c")
```

We compile the function on the command line using *R CMD SHLIB get64_c.c* to obtain the dynamic shared object (*get64_c.so* on our platform). Because of the *int64_t* argument, it is not possible to call this function with *.C()*. On the other hand, *.C64()* can interface this function when the second element of the *SIGNATURE* argument is set to *"int64"*.

```
R> dyn.load(paste0("get64_c", .Platform$dynlib.ext))
R> .C64("get64_c", SIGNATURE = c("double", "int64", "double"),
     input = x_long, index = 2^31, output = double(1))$output
[1] -1
```

In the call above, the function *.C64()* casts the argument *index* from *double* (the R representation of 64-bit integers) into an *int64_t* type vector before calling *get64_c()*, and back-casts it from *int64_t* to *double* afterwards.

By default the *INTENT* argument of *.C64()* is set to *"rw"* for all provided R objects. However, the C function *get64_c()* only reads (and not writes to) the arguments *input* and *index*, and only writes to (but not reads) the argument *output*. To avoid unnecessary copying of R objects we can modify the call as follows.

```
R> .C64("get64_c", SIGNATURE = c("double", "int64", "double"),
      INTENT = c("r", "r", "w"), input = x_long, index = 2^31,
       output = vector_dc("numeric", 1))$output
[1] -1
```

Note the usage of *numeric_dc()*, which implies that a double vector of length one is constructed by *.C64()*. This is computationally more efficient in case the *SIGNATURE* of that argument is *"int64"*. As shown by the performance measurements in Section S2 of the supplementary material, setting the correct *INTENT* reduces the time to pass large objects to compiled code significantly.

4. Conclusion and impact

About 3′200 R packages on CRAN interface compiled code and about 1′300 (40%) thereof use the FFI provided by R (as of 2018-01-09, see also Fig. 1). Using the FFI has one major advantage compared to more recent interfaces like the MIC and the R package Rcpp: It does not require the user to adapt the compiled code in an R specific way. This is convenient as it essentially reduces interfacing compiled code to calls of an R function. Thus, the FFI is well-suited to build high-level R environments on top of existing C, C++, and Fortran libraries. However, the FFI has severe limitations, which motivated the development of the proposed interface available in the R package dotCall64. dotCall64 enhances the FFI with the following innovations:

(A) the additional argument *SIGNATURE* ensures that the interfaced R objects are of the specified types
(B) options to avoid unnecessary copies of R objects
(C) support of long vectors
(D) casting mechanisms accounting for the R specific way of indexing long vectors
(E) parallel execution of time-consuming castings

The innovation (A) helps to avoid run-time errors and makes dotCall64 an interesting alternative for all current users of the FFI. However, the main benefits of using dotCall64 arise when dealing with large objects, which become increasingly important in data analysis. In that setting innovation (B) helps to avoid copying R objects that are read and not modified by the compiled code. This leads to considerable speed and memory gains as shown in the performance study presented in Section S2 of the supplementary material. Moreover, dotCall64 not only supports long vectors (C), but also provides automated double to 64-bit integer castings (D) parallelized with openMP (E). This facilitates using long R vectors jointly with compiled code; see Section S4 of the supplementary material for strategies to extend R packages with long vectors support. One prominent example where dotCall64 is used to extend an R package with long vector support is the sparse matrix algebra R package spam, which can now handle sparse matrices with more the $2^{31} - 1$ non-zero elements [25].

Acknowledgment

We acknowledge the support of the University of Zurich Research Priority Program (URPP) on "Global Change and Biodiversity".

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2018.06.002.

References

[1] Eubank RL, Kupresanin A. Statistical computing in C++ and R. Chapman & Hall/CRC; 2011.
[2] Chambers JM. Software for data analysis: Programming with R. Springer; 2008. http://dx.doi.org/10.1007/978-0-387-75936-4.
[3] R Core Team. Writing R extensions. Vienna, Austria: R Foundation for Statistical Computing; 2018 R version 3.5.0. URL http://cran.r-project.org/doc/manuals/R-exts.html.
[4] Wickham H. Advanced R. Chapman & Hall/CRC; 2014. http://dx.doi.org/10.1201/b17487.
[5] Sklyar O, Murdoch D, Smith M, Eddelbuettel D, François R, Soetaert K. inline: Inline C, C++, Fortran function calls from R. 2015 R package version 0.3.14. URL http://CRAN.R-project.org/package=inline.
[6] Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russell N, et al. Rcpp: Seamless R and C++ integration, R package version 0.12.14; 2017. URL https://CRAN.R-project.org/package=Rcpp.
[7] Eddelbuettel D, François R. Rcpp: Seamless R and C++ integration. J Stat Softw 2011;40(8):1–18. http://dx.doi.org/10.18637/jss.v040.i08.
[8] Eddelbuettel D. Seamless R and C++ integration with Rcpp. New York: Springer; 2013 URL http://www.rcpp.org/book/.
[9] Eddelbuettel D, François R, Bates D, Ni B. RcppArmadillo: Rcpp integration for the Armadillo templated linear algebra library, R package version 0.8.300.1.0. 2017 URL https://CRAN.R-project.org/package=RcppArmadillo.
[10] Eddelbuettel D, Sanderson C. RcppArmadillo: Accelerating R with high-performance C++ linear algebra. Comput Statist Data Anal 2014;71:1054–63. http://dx.doi.org/10.1016/j.csda.2013.02.005.
[11] Bates D, Eddelbuettel D, François R, Qiu Y, the authors of Eigen for the included version of Eigen. RcppEigen: Rcpp integration for the Eigen templated linear algebra library, R package version 0.3.3.3.1; 2017. URL https://CRAN.R-project.org/package=RcppEigen.
[12] Bates D, Eddelbuettel D. Fast and elegant numerical linear algebra using the RcppEigen package. J Stat Softw 2013;52(5):1–24. http://dx.doi.org/10.18637/jss.v052.i05.
[13] Allaire J, François R, Ushey K, Vandenbrouck G, Geelnard M, Intel. RcppParallel: Parallel programming tools for Rcpp R package version 4.3.20; 2016. URL https://CRAN.R-project.org/package=RcppParallel.
[14] François R, Ushey K, Chambers J. Rcpp11: R and C++11, R package version 3.1.2.0; 2014. URL https://CRAN.R-project.org/package=Rcpp11.
[15] Furrer R, Gerber F, Gerber D, Mösinger K. spam: SPArse Matrix, R package version 2.1-2; 2017. URL https://CRAN.R-project.org/package=spam.
[16] Furrer R, Sain SR. spam: A sparse matrix R package with emphasis on MCMC methods for Gaussian Markov random fields. J Stat Softw 2010;36(10):1–25. http://dx.doi.org/10.18637/jss.v036.i10.
[17] Gerber F, Furrer R. Pitfalls in the implementation of Bayesian hierarchical modeling of areal count data: An illustration using BYM and Leroux models. J Stat Softw 2015;63(1):1–32. http://dx.doi.org/10.18637/jss.v063.c01.
[18] Saad Y. SPARKIT: A basic tool kit for sparse matrix computations; 1994. URL http://www-users.cs.umn.edu/~saad/software/SPARSKIT/index.html.
[19] Oehlschlägel J. bit64: A S3 Class for Vectors of 64bit Integers, R package version 0.9-7; 2017. URL https://CRAN.R-project.org/package=bit64.
[20] Mösinger K, Gerber F, Furrer R. dotCall64: Enhanced Foreign Function Interface Supporting Long Vectors, R package version 0.9-5.2; 2018. URL https://CRAN.R-project.org/package=dotCall64.
[21] Wickham H. testthat: Get started with testing. R J 2011;3:5–10 URL http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.
[22] Dagum L, Menon R. OpenMP: An industry-standard API for shared-memory programming. IEEE Comput Sci Eng 1998;5(1):46–55. http://dx.doi.org/10.1109/99.660313.
[23] OpenMP architecture review board. OpenMP application program interface, version 4.5; 2016. URL http://www.openmp.org.
[24] Guest S. OpenMPController: Control number of OpenMP threads dynamically, R package version 0.2-5; 2017. URL http://CRAN.R-project.org/package=OpenMPController.
[25] Gerber F, Mösinger K, Furrer R. Extending R packages to support 64-bit compiled code: An illustration with spam64 and GIMMS NDVI_3g data. Comput Geosci 2017;104:109–19. http://dx.doi.org/10.1016/j.cageo.2016.11.015.